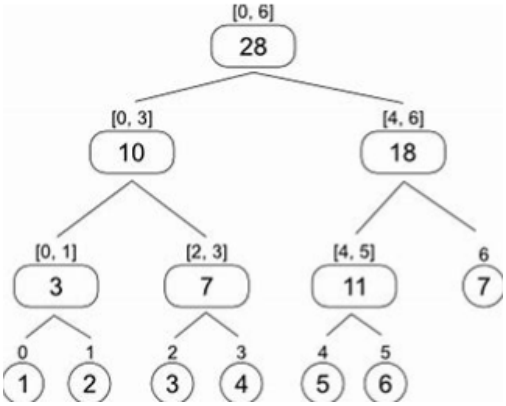
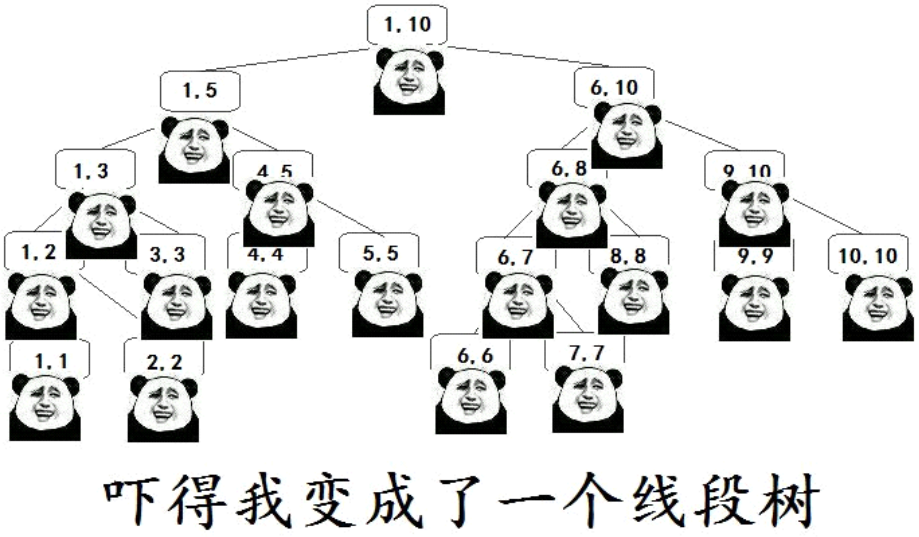
线段树

2019年4月12日 15:26

-
- 入门级线段树

1. 线段树Segment Tree
   1) 是基于分治思想的二叉树，用于在区间上统计信息
   2) 每个节点代表一个区间
   3) 唯一的根节点表示全域范围的区间[1,N]
   4) 每个非终端节点[l,r]的左子节点是[l,mid]，右子节点是[mid,r]，mid=(l+r)/2
   5) 

2. 父子2倍编号法数组存储
   1) 

   2) 因为最后一行不一定满，且n个叶子的满二叉树有2n-1个节点，所以数组长度需要4n来确保不越界
   3) 虽然直接乘以2就能得到左节点编号，但很难判断这个结点存不存在，最简单的判断法是给每个数据都配两个变量表示它代表的区间[l,r]
   4) struct segment_tree{
      int l,r;
      int data;
      }t[MN*4];
3. 建树（初始化赋值）
   1) 先递归找到单个数据a[i]应存在哪个叶节点，再将每个二度结点的值赋为两个子结点在操作f下的“和”

2) 参数1是节点编号，参数23是区间左右端点
3) void build(int p, int l, int r){
```
t[p].l=l, t[p].r=r;
if(l==r){
    t[p].data=a[l];
    return ;}
int mid=(l+r)/2;
build(p*2, l, mid);
build(p*2+1, mid+1, r);
t[p].data= f(t[p*2].data, t[p*2+1].data);}
```
4) 调用入口是build(1, 1, n);
5) p*2 == p<<1，p*2+1 == p<<1|1

4. 单点修改
1) 修改a[x]为v，并更新每个（祖）父节点
2) 需要先递归找到x下标对应树中叶结点的下标p，修改其值，再更新
3) void change(int p, int x, int v){
```
if(t[p].l==t[p].r){
    t[p].data=v;
    return ;}
int mid=(t[p].l+ t[p].r)/2;
if(x<=mid)
    change(p*2, x, v);
else
    change(p*2+1, x, v);
t[p].data=f(t[p*2].data, t[p*2+1].data);}
```
4) 调用入口是change(1, x, v);

5. 区间查询
1) 求区间[l,r]的f运算的累积结果
2) 若当前下标p代表的区间被lr完全覆盖住了，直接返回其值
3) 否则说明当前代表区间有一部分被lr覆盖了，需调查是左子树还是右子树
4) 麻烦的是可能左右子树各有一部分重叠，此时需返回左右子树f运算的结果
5) 如果f运算有“幺元”的话就好办了，比如求最大值运算在绝对值较小的实数域中赋初值为-(1<<30)，加法运算在实数域中赋初值为0，乘法运算在实数域中赋初值为1，再对左右各求一次f运算即可
6) int ask(int p, int l, int r){
```
if(t[p].l>=l&& t[p].r<=r)
    return t[p].data;
int mid= t[p].l+ t[p].r >>1;
int val=0;          //f运算的其他对应幺元
if(l<=mid)
    val=f(val, ask(p*2, l, r));
if(r>mid)           //r>=mid+1
    val=f(val, ask(p*2+1, l, r));
return val; }
```

6. 区间查询的时间复杂度分析（令pl，pr表示当前节点对应区间端点）
1) l<= pl<= pr<= r时左右子树和区间本身被完全覆盖，直接return
2) pl< l<= pr<= r时左子树一定没有被完全覆盖
   i. l> mid时右子树也没有被完全覆盖，只递归右子树
   ii. l<= mid时右子树被完全覆盖，一次调用直接return
3) l<= pl<= r< pr时右子树一定没有被完全覆盖

i. r<= mid时左子树也没有被完全覆盖，只递归左子树
ii. r> mid时左子树被完全覆盖，一次调用直接return

4) pl< l<= r< pr时左右子树都一定没有被完全覆盖
   i. r<= mid或l> mid时分别只递归左右子树
   ii. r> mid且l<= mid时左右子树都需要递归
5) 只有当l！=r时才会在4）ii复杂递归两次以上，且这个情况只会发生一次，综合考虑，时间复杂度约为O（2logN）=O（logN）

7. 带延迟标记的区间修改
   1) 适用于要给区间增加d，求区间和的情况
   2) 当区间结点整个被覆盖时，可以通过给该结点值+=区间长*d，直接返回
   3) 此时需要给该结点打标记，表示虽然当前结点的值都更新好了，但+d没有更新给叶结点，下次需要往下传
   4) 在change时判断完不是完全覆盖后，立刻判断是否需要向下一层传标记
   5) void spread(int p){ //往下一层传延迟标记

```
if(tag[p]){
    d[p*2] += tag[p]* (r[p*2]- l[p*2]+ 1);
    d[p*2+1] += tag[p]* (r[p*2+1]- l[p*2+1]+ 1);
    tag[p*2] += tag[p];
    tag[p*2+1] += tag[p];
    tag[p] = 0;}}
```

   6) void add(int p, int L, int R, int dx){ //对[L,R]+=dx

```
if(L<=l[p] && r[p]<=R){
    d[p] += dx* (r[p]- l[p]+ 1);
    tag[p] += dx;
    return ;}
spread(p);
int mid= (l[p]+ r[p])/2;
if(R>mid)
    add(p*2+1,L,R,dx);
if(L<=mid)
    add(p*2,L,R,dx);
d[p]= d[p*2]+ d[p*2+1];}
```

8. zkw线段树
   1) 利用完全二叉树的下标存法，多浪费一点下标空间，实现O（1）查询叶结点
   2) 之后用类似二叉堆找根的方法，不断右移下标，实现向上更新

-
- 例

```
struct ST{//区间修最值
    struct node{
        int l, r;
        int m, M, tg;
    }t[MN<<2];
    void up(int p){
        t[p].m = min(t[p<<1].m, t[p<<1|1].m);
        t[p].M = max(t[p<<1].M, t[p<<1|1].M);
    }
    void down(int p){
        int &tg = t[p].tg;
        t[p<<1].m += tg;
        t[p<<1].M += tg;
        t[p<<1].tg += tg;
```

```
            t[p<<1|1].m += tg;
            t[p<<1|1].M += tg;
            t[p<<1|1].tg += tg;
            tg = 0;
        }
        void build(int l=1, int r=n, int p=1){
//printf("building:%d(%d~%d)\n",p,l,r);
            t[p].l=l, t[p].r=r, t[p].tg=0;
            if(l==r) return t[p].m=0, t[p].M=0, void(0);
            int mid=l+r>>1;
            build(l, mid, p<<1);
            build(mid+1, r, p<<1|1);
            up(p);
        }
        void add(int L, int R, int V, int p=1){
//printf("adding:%d(%d~%d) target:[%d~%d]+= %d\n",p,t[p].l,t[p].r,L,R,V);
            int l=t[p].l, r=t[p].r;
            if(L<=l && r<=R) return t[p].m+=V, t[p].M+=V, t[p].tg+=V, void(0);
            if(t[p].tg) down(p);
            int mid=l+r>>1;
            if(L<=mid) add(L, R, V, p<<1);
            if(mid<R) add(L, R, V, p<<1|1);
            up(p);
        }
        int m(int L, int R, int p=1){
//printf("askingm:%d(%d~%d) target:[%d~%d]\n",p,t[p].l,t[p].r,L,R);
            int l=t[p].l, r=t[p].r;
            if(t[p].tg) down(p);
            if(L<=l && r<=R) return t[p].m;
            int mid=l+r>>1;
            int m1 = 1<<29, m2 = 1<<29;
            if(L<=mid) m1 = m(L, R, p<<1);
            if(mid<R) m2 = m(L, R, p<<1|1);
            up(p);
            return min(m1,m2);
        }
        int M(int L, int R, int p=1){
//printf("askingM:%d(%d~%d) target:[%d~%d]\n",p,t[p].l,t[p].r,L,R);
            int l=t[p].l, r=t[p].r;
            if(t[p].tg) down(p);
            if(L<=l && r<=R) return t[p].M;
            int mid=l+r>>1;
            int M1 = -(1<<29), M2 = -(1<<29);
            if(L<=mid) M1 = M(L, R, p<<1);
            if(mid<R) M2 = M(L, R, p<<1|1);
            up(p);
            return max(M1,M2);
        }
}st;

struct ST{//扫描线
    struct node{ //每个结点代表一个扫描线区间，通过函数参数lr判断其纵坐标
        int l,r;     //当前区间对应的左右端点
        int len,cnt; //区间内被覆盖的区间的总长，当前区间被整个覆盖的次数
    }t[MN<<4];   //结点编号空开0
    void up(int p){     //更新结点p内的区间长度
        if(t[p].cnt) t[p].len= oy[t[p].r+1]- oy[t[p].l];
        else t[p].len= t[p<<1].len+ t[p<<1|1].len;
    }
    void build(int p, int l, int r){ //初始化区间p
//printf("building:%d(%d~%d)\n",p,l,r);
        t[p].l=l, t[p].r=r;
        if(l==r) return t[p].len=0/*oy[r+1]-oy[r]*/, t[p].cnt= 0, void(0);
        int mid= l+r >>1;
```

```
          build(p<<1, l, mid);
          build(p<<1|1, mid+1, r);
//           up(p);
     }
     void add(int p, int L, int R, int v){  //给区间[L,R]+=v
//printf("adding:%d(%d~%d) target:[%d~%d]+= %d\n",p,t[p].l,t[p].r,L,R,v);
          int l=t[p].l, r=t[p].r;
          if(L<=l && r<=R) return t[p].cnt+=v, up(p), void(0);
          int mid= l+r >>1;
          if(L<=mid) add(p<<1, L, R, v);
          if(mid<R) add(p<<1|1, L, R, v);
          up(p);
//printf(" %dlen now:%d\n",p,t[p].len);
     }
}sl;  //扫描线，扫描线总长存在sl.t[1].len中
int oy[MN<<2];      //离散化y的数组，oy[i]=第i小的y，空开下标0
int newn;    //去重后的y的数量
int lb(int y){      //在排序后的oy数组中二分查找y的下标
     return lower_bound(oy+1,oy+newn+1,y) -oy;
}
struct Line{
     int x,y,Y,f; //横坐标，纵坐标的最大最小值，是否左边界
     bool operator<(const Line&t)const{
          return x!=t.x? x<t.x: f>t.f;
     }
}e[MN<<2];
     int n; scanf("%d",&n);
     for__(i,1,n){
          int l,d,r,u;
          scanf("%d%d%d%d",&l,&d,&r,&u);
          e[i]={ l,d,u,1 };
          e[i+n]={ r,d,u,-1 };
          oy[i]= d;
          oy[i+n]= u;
     }
     int n2=n<<1;
     sort(oy+1,oy+n2+1);
     newn= unique(oy+1,oy+n2+1)-oy-1;
     oy[newn+1]=oy[newn];      //最高的y上再多加一个y，防止爆
//for__(i,0,newn) printf("%d %d\n",i,oy[i]);
     sort(e+1,e+n2+1);
     e[n2+1].x=e[n2].x;  //最后一条扫描线的后面加一条
     sl.build(1,1,newn-1);
     ll ans=0;
     for__(i,1,n2){
          sl.add(1,lb(e[i].y),lb(e[i].Y)-1,e[i].f);
          ll len= sl.t[1].len;
//printf("len now:%lld  width now:%d\n",len,e[i+1].x-e[i].x);
          ans+= len* (e[i+1].x-e[i].x);
     }
     printf("%lld\n",ans);
```

9. 区间最值子区间的端点，多解时输出字典序最小的（USST集训4H）
   1) 需要传递的值比较多，可以让需要向上传递的函数都返回一个结点作为右值

```
struct ST{
     int l,r;     //本身区间端点
     int ml,mr;   //最大子区间端点
     int lr;            //最大前缀区间右端点
     int rl;            //最大后缀区间左端点
```

```
    ll s,ms,ls,rs;      //总区间和，最大区间和，最大前缀和，最大后缀和
}t[MN*4];
inline ST up(ST pl, ST pr){      //将左右子结点更新给父节点，返回一个结点作为右值
    ST rt= pl;   //{pl.l, pr.r, pl.ml, pl.mr, pl.lr, pl.rl, pl.s+pr.s, pl.ms, pl.ls, pl.rs+pr.s};
    rt.r= pr.r;
    rt.s+= pr.s;
    rt.rs+= pr.s;
//题目要求输出字典序小的编号，所以初值都赋为与pl有关的，之后再判断需不需要改成右边的
    if(rt.ls < pl.s+pr.ls)    //前缀需向右扩展
        rt.ls= pl.s+pr.ls,
        rt.lr= pr.lr;
    if(rt.rs < pr.rs)  //后缀不应向左扩展
        rt.rs= pr.rs,
        rt.rl= pr.rl;
    if(rt.ms < pl.rs+pr.ls)   //最大区间被分在两个子区间
        rt.ms= pl.rs+pr.ls,
        rt.ml= pl.rl,
        rt.mr= pr.lr;
    if(rt.ms < pr.ms)  //最大区间在右子结点，注意这个必须在上一个if之后判断
        rt.ms= pr.ms,
        rt.ml= pr.ml,
        rt.mr= pr.mr;
    return rt;}
void build(int p,int l,int r){
    if(l==r){
        t[p].ml= t[p].mr= t[p].l= t[p].lr= t[p].rl= t[p].r= l;
        t[p].s= t[p].ls= t[p].ms= t[p].rs= a[l];
//cout<<"!built"<<l<<"~"<<r<<" M:"<<t[p].ml<<"~"<<t[p].mr<<'\n';
        return;}
    int mid= l+r >>1;
    int pl= p<<1;
    int pr= pl|1;
    build(pl, l, mid);
    build(pr, mid+1, r);
    t[p]= up(t[pl],t[pr]);}
ST ask(int p,int l,int r){//返回存有想要的信息的结点
//cout<<"asking"<<l<<"~"<<r<<" in t:"<<t[p].l<<"~"<<t[p].r<<'\n';
    if(t[p].l>=l && t[p].r<=r)//被完全覆盖
        return t[p];
    int pl= p<<1;
    int pr= pl|1;
    int mid= t[p].l+ t[p].r >>1;
    if(mid>=l && mid<r) //两边各取一半
        return up(ask(pl,l,r), ask(pr,l,r));
    else if(mid>=l)           //只取左
        return ask(pl,l,r);
    else if(mid<r)            //只取右
        return ask(pr,l,r);}
```

10. 统计只有两个数的最长非降子序列，带区间互换（CF145E）
    1) 统计区间最长4，最长7，最长前半4后半7，最长前半7后半4
    2) 可以用类似上一题那样的分类讨论合并
    3) 每次区间互换时，只需换4和7，47和74

4) 区间修懒标为奇数时才需要下传，偶数时其实不用传

```
struct ST{//区间修，讨论各种最长序列的最值
    struct node{
        int l,r;
        int c4,c7,c47,c74;  //各种序列的最长长度
        int tg;       //逆转懒标，因为转两次就相当于没转，所以它的加法是模2的
    }t[MN<<2];   //结点编号空开0
    void up(int p){     //更新结点p内的区间长度
        node &tp=t[p], &tl=t[p<<1], &tr=t[p<<1|1];
        tp.c4= tl.c4+ tr.c4;
        tp.c7= tl.c7+ tr.c7;
        tp.c47= max(tl.c4+tr.c47, tl.c47+tr.c7);
        tp.c47= max(tp.c47, tl.c4+tr.c7);
        tp.c74= max(tl.c7+tr.c74, tl.c74+tr.c4);
        tp.c74= max(tp.c74, tl.c7+tr.c4);
    }
    void sw(int p){     //交换p结点的4和7
        node &tp=t[p];
        tp.tg^=1;     //多改了一次
        swap(tp.c4,tp.c7);
        swap(tp.c47,tp.c74);
    }
    void down(int p){   //转了奇数次的话，下传懒标
        int &tg=t[p].tg;
        if(!tg) return;
        sw(p<<1);
        sw(p<<1|1);
        tg=0;
    }
    void build(int p, int l, int r){ //初始化区间p
//printf("building:%d(%d~%d)\n",p,l,r);
        node &tp=t[p];
        tp.l=l, tp.r=r, tp.tg= tp.c4= tp.c7= tp.c47= tp.c74= 0;
        if(l==r) return ++(s[l]=='4'? tp.c4: tp.c7), void(0);
        int mid= l+r >>1;
        build(p<<1, l, mid);
        build(p<<1|1, mid+1, r);
        up(p);
    }
    void change(int p, int L, int R){       //给区间[L,R]的4和7互换
//printf("adding:%d(%d~%d) target:[%d~%d]\n",p,t[p].l,t[p].r,L,R);
        node &tp=t[p];
        int l=tp.l, r=tp.r;
        if(L<=l && r<=R) return sw(p), void(0);
        if(tp.tg) down(p);
        int mid= l+r >>1;
        if(L<=mid) change(p<<1, L, R);
        if(mid<R) change(p<<1|1, L, R);
        up(p);
    }
    ll ask(int p, int L, int R){     //询问区间[L,R]的最长递增子序列
//printf("asking:%d(%d~%d) target:[%d~%d]\n",p,t[p].l,t[p].r,L,R);
        node &tp=t[p];
        int l=tp.l, r=tp.r;
        if(tp.tg) down(p);
        if(L<=l && r<=R) return max(tp.c47, max(tp.c4, tp.c7));
        int mid= l+r >>1;
        ll m1=0, m2=0;
        if(L<=mid) m1=ask(p<<1,L,R);
```

```
            if(mid<R) m2=ask(p<<1|1,L,R);
            up(p);
            return max(m1,m2);
        }
    }st;
```

11. 统计平行于坐标轴方向的直线能与几根线段有交点

1) 即把每根线段投影到x轴和y轴上，给每个投影点的数量+1

2)
```
const int MN = 4*1000005;
stack< pair<int,int> > sx,sy;
struct segTree{
    int l[MN], r[MN];
    int d[MN];
    int tag[MN];     //延迟标记
    void build(int p, int pl, int pr){    //下标p==[pl,pr]
        l[p]= pl;
        r[p]= pr;
        if(pl==pr){
            d[p]= 0;
            return ;}
        int mid= (pl+pr)/2;
        build(p*2, pl, mid);
        build(p*2+1 ,mid+1, pr);
        d[p]= d[p*2]+ d[p*2+1];}
    void spread(int p){                    //往下一层传延迟标记
        if(tag[p]){
            d[p*2] += tag[p]* (r[p*2]- l[p*2]+ 1);
            d[p*2+1] += tag[p]* (r[p*2+1]- l[p*2+1]+ 1);
            tag[p*2] += tag[p];
            tag[p*2+1] += tag[p];
            tag[p] = 0;}}
    void add(int p, int L, int R, int dx){          //对[L,R]+=dx
        if(L<=l[p] && r[p]<=R){
            d[p] += dx* (r[p]- l[p]+ 1);
            tag[p] += dx;
            return ;}
        spread(p);
        int mid= (l[p]+ r[p])/2;
        if(R>mid)
            add(p*2+1,L,R,dx);
        if(L<=mid)
            add(p*2,L,R,dx);
        d[p]= d[p*2]+ d[p*2+1];}
    int ask(int p, int L, int R){           //求[L,R]的和
        if(L<=l[p] && r[p]<=R){
            return d[p];}
        spread(p);
        int ret= 0;
        int mid= (l[p] + r[p])/2;
        if(R>mid)
            ret+= ask(p*2+1,L,R);
        if(L<=mid)
            ret+= ask(p*2,L,R);
        return ret;}
}x,y;
```

3)
```
ios::sync_with_stdio(0);
x.build(1,1,1000000);
y.build(1,1,1000000);
cin>>n;
int op,xl,yl,xr,yr;
for_(i,0,n){
	cin>>op;
	switch(op){
		case 1:
			cin>>xl>>yl>>xr>>yr;
			if(xl>xr)
				swap(xl,xr);
			if(yl>yr)
				swap(yl,yr);
			x.add(1,xl,xr,1);
			y.add(1,yl,yr,1);
			sx.push(make_pair(xl,xr));
			sy.push(make_pair(yl,yr));
			break;
		case 2:
			cin>>xl;
			cout<<x.ask(1,xl,xl)<<'\n';
			break;
		case 3:
			cin>>yl;
			cout<<y.ask(1,yl,yl)<<'\n';
			break;
		default:
			if(!sx.empty()){
				xl= sx.top().first;
				xr= sx.top().second;
				sx.pop();
				x.add(1,xl,xr,-1);}
			if(!sy.empty()){
				yl= sy.top().first;
				yr= sy.top().second;
				sy.pop();
				y.add(1,yl,yr,-1);}}}
```